Extracted from:

Rust Brain Teasers

Exercise Your Mind

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Rust Brain Teasers

Exercise Your Mind

Herbert Wolverson

edited by Tammy Coron

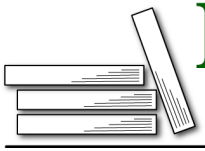
Rust Brain Teasers

Exercise Your Mind

Herbert Wolverson

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Tammy Coron
Copy Editor: L. Sakhi MacMillan
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-680509-17-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2022

To Henry, my loyal canine coding companion
of thirteen years—who sadly didn't live to see
the book's release.

Puzzle 14

Structure Sizing

structure_sizing/src/main.rs

```
use std::mem::size_of;

struct VeryImportantMessage {
    _message_type: u8,
    _destination: u16
}

fn main() {
    println!(
        "VeryImportantMessage occupies {} bytes.",
        size_of::<VeryImportantMessage>()
    );
}
```

Guess the Output

Try to guess what the output is before moving to the next page.

The program will display the following output:

```
VeryImportantMessage occupies 4 bytes.
```

Discussion

`_message_type` and `_destination` are sized as you would expect, occupying 1 and 2 bytes of memory, respectively. So, why does `VeryImportantMessage` occupy 4 bytes of memory?

By default, Rust makes two promises about the in-memory representation of your structures:

- Structures may be sized differently than their contents for performance reasons.
- Structures may store data in a different order internally than you specified if the optimizer believes it will aid performance.

Most modern CPUs align data on 32-bit boundaries in memory and cache. Accessing 8 bits (one byte) or 16 bits (two bytes) is fast because the CPU provides primitives to do so, and the structures can be packed along 32-bit boundaries.

A 24-bit (3 byte) structure doesn't naturally align to a 32-bit memory map, so by default, Rust wastes 8 bits of memory per `struct` to ensure fast access to the structure in your computer's memory. This behavior is especially helpful when you're dealing with arrays or other contiguous blocks of 3-byte structures because every other structure would start at the 24th bit of a 32-bit block, reducing cache and read efficiency.

Sometimes this behavior can cause problems, though. For example:

- If you're storing a very large number of 24-bit structures, wasting a byte per structure might exceed your memory allocation—especially on embedded systems.
- If you're interoperating with another language, that language may expect your structures to be *exactly* 24 bits in size. Conversely, a server written in Rust may add padding to structures—surprising the client with padding bytes.

- Likewise, if you're interoperating with other languages, letting Rust rearrange your data in memory can cause bizarre problems when passing data to and from the other language.

Constraining Rust's Optimizer

You can turn off both of Rust's structure optimizations using a decoration named #[repr()], which gives you control over how a struct is represented in memory:

- #[repr(C)] declares that you require interoperability with the C language. Rust won't rearrange the content of your structure.
- #[repr(packed)] tells Rust not to waste space on your structure. This can carry a small performance penalty but guarantees that structures are exactly the right size.

You can combine these decorations. For example, a structure decorated with #[repr(C, packed)] won't rearrange or pad your structure:

```
#[repr(C, packed)]
struct ReallyThreeBytes {
    a: u8,
    b: u16
}

fn main() {
    println!(
      "ReallyThreeBytes occupies {} bytes.",
      size_of::<ReallyThreeBytes>()
    );
}
```

This code prints:

```
ReallyThreeBytes occupies 3 bytes.
```

Further Reading

repr(Rust)
: https://doc.rust-lang.org/nomicon/repr-rust.html

Type Layout
: https://doc.rust-lang.org/reference/type-layout.html

Layout of structs and tuples
: https://rust-lang.github.io/unsafe-code-guidelines/layout/structs-and-tuples.html